Out-of-Core Real-Time Visualization of Massive 3D Point Clouds

Rico Richter[*]
University of Potsdam
Hasso-Plattner-Institute

Jürgen Döllner[†]
University of Potsdam
Hasso-Plattner-Institute

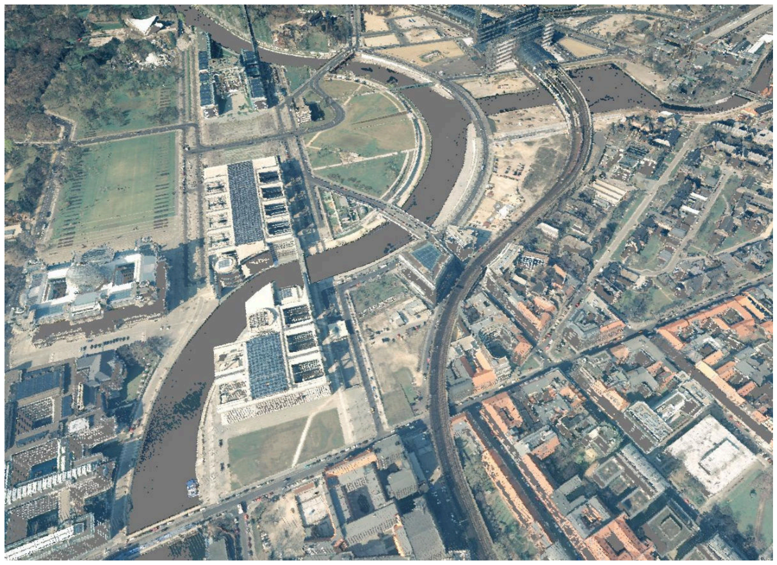


Figure 1: *Point-based rendering of a massive 3D point cloud with approximately 5 billion points captured with an airborne laser scanner.*

Abstract

This paper presents a point-based rendering approach to visualize massive sets of 3D points in real-time. In many disciplines such as architecture, engineering, and archeology LiDAR technology is used to capture sites and landscapes; the resulting massive 3D point clouds pose challenges for traditional storage, processing, and presentation techniques. The available hardware resources of CPU and GPU are limited, and the 3D point cloud data exceeds available memory size in general. Hence out-of-core strategies are required to overcome the limit of memory. We discuss concepts and implementations of rendering algorithms and interaction techniques that make out-of-core real-time visualization and exploration of massive 3D point clouds feasible. We demonstrate with our implementation real-time visualization of arbitrarily sized 3D point clouds with current PC hardware using a spatial data structure in combination with a point-based rendering algorithm. A rendering front is used to increase the performance taking into account user interaction as well as available hardware resources. Furthermore, we evaluate our approach, describe its characteristics, and report on applications.

Keywords: out-of-core visualization, point-based rendering, 3D point clouds, LiDAR

[*]e-mail: rico.richter@hpi.uni-potsdam.de
[†]juergen.doellner@hpi.uni-potsdam.de

1 Introduction

LiDAR (Light Detection And Ranging) scanning technology facilitates fast capturing of physical objects with high sampling rates and typically generates billions of 3D points for a target region. In the following a vast amount of 3D points exceeds typical main memory capacity is called *massive 3D point cloud*. Popular LiDAR capturing methods include airborne laser scanning with aircrafts and terrestrial laser scanning with portable scanners. The resulting 3D point clouds reflect the geometry of the reality by discrete, point-based data. They are used as base data to create and update contents of 3D geovirtual environments (GeoVEs), such as terrain, building, and site models. A large number of building and surface reconstruction algorithms are based on 3D point clouds to generate models for GeoVEs [Poullis and You 2009]. Typically reconstruction applications thin out and resample points into a grid, because many reconstruction algorithms are not able to handle massive 3D point clouds directly. Applications and systems frequently demand to compare an entire 3D point cloud within a GeoVE. This way tasks such as error correction, editing, and quality control can be facilitate by the interactive examination of the 3D point cloud. Due to their massivity straightforward 3D rendering approaches are not feasible. This motivates our approach of out-of-core 3D viewing of massive LiDAR datasets, which may contain billions of 3D points. The main challenge is to guarantee both, high frame rates during user interaction and high rendering quality.

In our approach, a preprocessing step converts a given 3D point cloud into a spatial data structure to determine the point representation of an object at an arbitrary level of detail during runtime. The traversal of the datastructure mainly depends on user interactivity. It also provides memory management to adaptively load parts of the structure from external memory. Thus, it is possible to interactively visualize 3D point clouds of arbitrary size. The visualization of 3D point clouds is implemented by point-based rendering. The contributions of this paper are strategies to render massive 3D point clouds on standard consumer hardware. The main contribution is an algorithm that uses the frame-to-frame coherence to traversal the datastructure. The rendering algorithm is adapting to the user interaction, memory resources, and hardware performance. Scalability and real-time capabilities of the technique are analyzed in performance tests.

This paper is organized as follows: Section 2 gives an overview about existing techniques and related work. Section 3 presents the 3D point cloud rendering algorithms and introduces the multiresolution data structure as well as strategies to render massive 3D point clouds. In Section 4 we present an example dataset with rendering results. Section 5 gives conclusions and describes future work.

2 Related Work

Various point-based multiresolution rendering techniques became popular in recent years. Surfels [Pfister et al. 2000] is based on an octree storing sample points. QSplat [Levoy and Rusinkiewicz 2000] is based on a bounding sphere hierarchy to provide splats for the rendering process; the tree structure is based on a kd-tree containing positions, normals and colors. In our approach we use an octree, and some of the rendering principles of QSplat have been adapted.

Many solutions have been proposed for point-based rendering, for an overview see Sains et al. [2004]. Most of them focus on efficient rendering and display quality for surfaces [Zwicker et al. 2001; Botsch and Kobbelt 2003]. Xu et al. [2004] have investigated non-photorealistic rendering techniques for 3D point clouds. Dachsbacher et al. [2003] have introduced sequential point trees to efficiently render 3D point clouds by evaluating the data structure on the GPU. This approach is limited by the available GPU memory and is not suited for huge LiDAR data.

In general, assumptions about existing normals for each point and sampling densities are made. Normal vectors for each point can be estimated to improve the rendering quality [Dey et al. 2005]. High sampled models are rendered with splatted point primitives oriented on the existing normals for each point [Pfister et al. 2000] to perform shading [Botsch et al. 2005]. The focus of our work, however, is not primarily to reach high quality rendering results for surfaces. We focus on the quantity to render huge 3D point clouds with out-of-core techniques that enable fast exploration of the data. XSplat [Pajarola et al. 2005] is a rendering system based on [Dachsbacher et al. 2003] with out-of-core strategies to render massive 3D point clouds. It concentrates on display quality and has been designed to render sampled models with continuous surfaces. We reduce the assumptions for input data, because our data generally results from LiDAR scans and, hence, contains only position and color information, i.e., we treat 3D point clouds as non-continuous surfaces.

Pauly et al. [2003] have introduced a modelling framework for point-based geometries. A datastructure to enable surface modification at different scales was introduced by Pauly et al. [2006]. Rendering systems with out-of-core data structures to edit point clouds have been introduced by Zwicker et al. [2002] Wand et al. [2007], Kreylos et al. [2008] and Schneiblauer et al. [2009]. The datastructures are designed to edit, insert, and remove points during the rendering process. In our approach massive 3D point clouds have to be handled while no editing is supported, which allows us to keep the datastructure compact.

Wimmer et al. [2006] have presented a rendering approach that does not assume any sampling density or availability of normals for rendering of LiDAR data. The rendering of unprocessed 3D point clouds is based on memory-optimized sequential point trees and has out-of-core strategies to handle huge point clouds. Our approach has the same terms for the input data, but differs in the used rendering strategies. We do not use the GPU-based sequential point tree approach that has been first introduced by Dachsbacher et al. [2003] and adapted by the Instant Points rendering system of Wimmer et al.[2006]. Our approach is CPU-based and assumes some basic strategies from QSplat [Levoy and Rusinkiewicz 2000] to perform efficient culling. We use the GPU memory only to store points that are displayed. So we are able to use the limited GPU memory efficiently.

Bettio et al. [2009] have introduced a client-server framework to render large point-based 3D models, which can perform different measurements. In contrast to our system the used data structure is based on Layered Point Clouds from Gobbetti [2004]. To reduce the complexity of 3D point clouds Boubekeur et al. [2005] used normal-mapped polygons as approximation.

Some other approaches propose hybrid rendering to increase frame rates by replacing point sampled areas with meshes [Wahl et al. 2005]. Rendering of triangle-based meshes assumes a simplification of the mesh [Lindstrom 2000]. In contrast to massive 3D point clouds connectivity information for meshes must be stored. Most solutions use a progressive mesh representation for storing and transmitting arbitrary triangle meshes [Hoppe 1996]. Out-of-core techniques to render huge triangle meshes have been well investigated [Gobbetti et al. 2008; Callieri et al. 2008; Wald et al. 2004].

Streaming solutions for meshes [Isenburg and Lindstrom 2005] and 3D point clouds enable a fast transfer of data into main memory [Pajarola 2005] but are limited for interactive visualizations. In general LiDAR data does not contain color information after capturing. This information can be added by using multiple images [Abdelhafiz and Niemeier 2006] or projected videos during the rendering process [Zhao et al. 2005]. In our system, colors from aerial images are assigned to 3D points.

3 Massive 3D Point Cloud Rendering

This section describes the system to create and render the multiresolution datastructure. The system consists of different modules, shown in Fig. 2. 3D point clouds from different sources and possibly in different formats can be handled. The input data is transformed into a uniform coordinate system and all points are standardized to a predefined precision. LiDAR data contains informa-

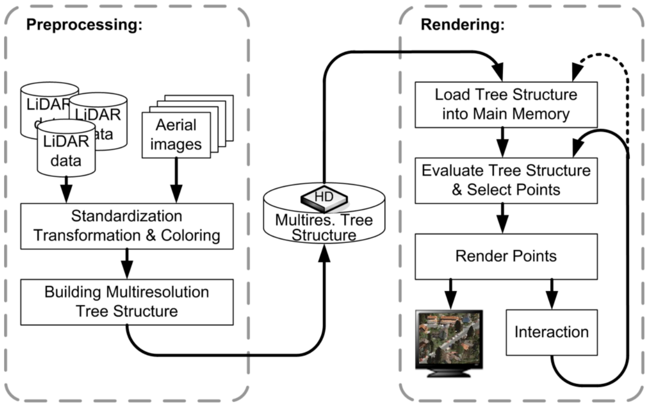

Preprocessing:
LiDAR data
LiDAR data
LiDAR data
Aerial images
Standardization Transformation & Coloring
Building Multiresolution Tree Structure
HD
Multires. Tree Structure
Rendering:
Load Tree Structure into Main Memory
Evaluate Tree Structure & Select Points
Render Points
Interaction

Figure 2: *System overview containing components for preprocessing and rendering of massive 3D point clouds.*

tion about position and optionally reflection density or a color value for each point. If only position information are available aerial images can be used to color LiDAR data from airborne laser scans [Abdelhafiz et al. 2005]. The explanation of techniques to color massive 3D point clouds are beyond the scope of this paper. After standardizing point data an out-of-core algorithm computes the multiresolution data structure, which is stored on the external memory. This structure is used to extract any level of detail of the point data for the rendering process depending on user interaction and available hardware resources. If the tree structure exceeds the available main memory resources only upper parts of the tree structure are loaded for rendering. If parts with more details of the LiDAR scan are needed during interaction and exploration, relevant parts are loaded adaptively from the external memory at runtime. Details for preprocessing and creating the multiresolution data structure are given in Section 3.1; the rendering strategies are explained in Section 3.2.

3.1 Multiresolution Out-of-Core Data Structure

An octree is used to prepare and store the 3D point cloud for the rendering system. It is well suited to split the entire dataset in manageable parts in a short time. In contrast to a kd-tree, there is no need to sort the entire dataset along the longest axis for each tree level. This is time-consuming for massive 3D point clouds. The use of an octree leads to faster preprocessing times and fulfills the requirement for the rendering algorithm for fast traversal. The tree structure represents a bounding sphere hierarchy and is illustrated in Fig. 3 with a small dataset for different tree levels. Each node stores information about position, color, radius, and the tree structure. Leaf nodes store also a list with a maximum number of points. We use bounding spheres to represent inner nodes of the tree because the calculation and culling can be performed fast. Bounding boxes or oriented disks are more precise for planar groups of points but decelerate the rendering, as described in Section 3.2. Figure 4 shows the size of the nodes in different tree levels.

The structure is calculated and serialized in a preprocessing step according to QSplat [Levoy and Rusinkiewicz 2000]. Each node in the tree is a generalization of its child nodes. All 3D points of the input LiDAR data are stored in the leaf nodes. In contrast to the implementation of QSplat more than one point is stored in the leaf nodes to prevent deep trees. The point capacity for leaf nodes affects the total number of nodes in the tree, which has an impact on the main memory requirements and the rendering quality of the tree structure. A low capacity allows the rendering algorithm fine-grained and continuous refinements of the 3D point cloud during rendering if leaf nodes are reached. Also frustum culling can be performed more accurately for smaller leave nodes. If the number of points in a leaf node is higher, a lower number of tree nodes in the tree structure is necessary and more data can be stored in main memory when dealing with massive 3D point clouds. Also the evaluation of the tree structure can be performed faster which decreases the time of the rendering process. For this reason our tree structure stores more than one point in the leaf nodes of the tree in contrast to the implementation of QSplat. Figure 3 illustrates the tree structure with five tree levels and the properties of both node types.

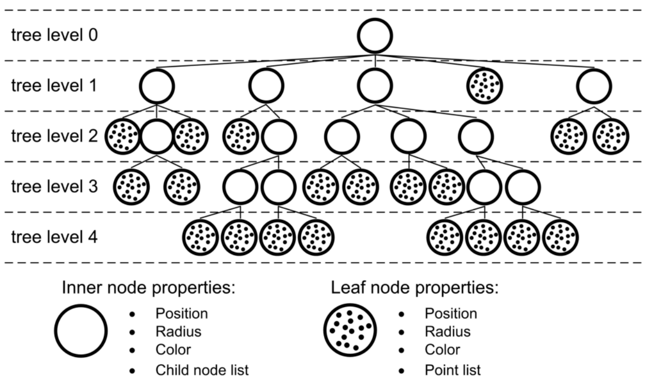


Figure 3: *Illustration of the octree structure with properties of inner and leaf nodes.*

While creating the tree structure every point is assigned to a corresponding node of the tree. If the maximum capacity of a leaf node is reached, the leaf node is subdivided into eight cells and it becomes a representation of its new child nodes. All points are assigned to the new child nodes. After assigning all the points to the tree, properties of inner nodes are calculated bottom up. The mean position and color of the children is calculated. Also the radius to create a sphere that encloses all nodes and points in lower tree levels is calculated. Octree cells without points are not stored in the tree structure, so each inner node has between two and eight child nodes depending on the arrangement of the LiDAR points in the 3D point cloud.

The computation is done by an out-of-core preprocessing step to handle 3D point clouds of any size. For this reason the size of the input LiDAR point cloud is calculated and compared with the available main memory. If the 3D point cloud exceeds available main memory, it is divided into eight sub-clouds depending on the dividing rules of an octree. Each sub-cloud containing 3D points is processed again until the size of the sub-cloud fits into the main memory. If the point file for a corresponding octree cell is small enough, all points are converted into the octree structure in the main memory. Once all sub-clouds are calculated they can be merged to a larger 3D point cloud tree file.

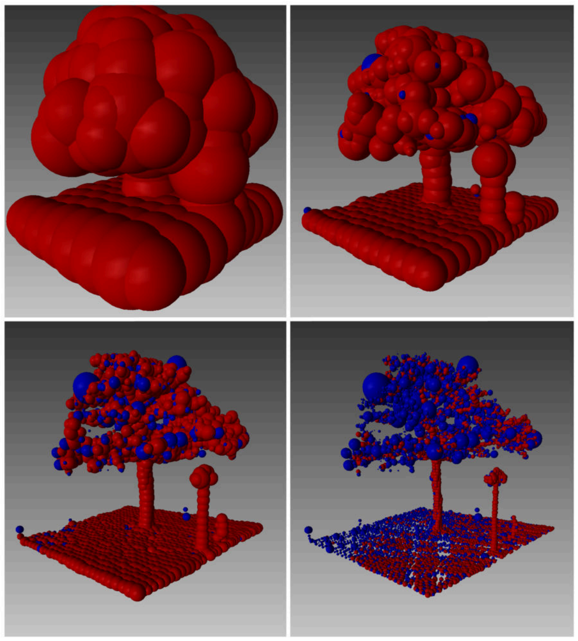


Figure 4: *Illustration of the tree structure with bounding spheres for tree levels 3–6. Each sphere represents a tree node and encloses all child nodes of the presented node. Inner nodes are illustrated with red und leaf nodes with blue color.*

At the end of the preprocessing, one treefile containing the entire octree is formed. This procedure to create the spatial data structure is illustrated in Fig. 5. To serialize octrees the pointer structure between parent and child nodes is mapped to file position values. These values enable fast access and switches between main memory and external memory during rendering. Each node can be identified by a 64 bit integer value. For this reason, the tree size is only limited by the size of the external memory. For rendering of huge tree structures, only the first few levels of the tree are loaded from external memory into main memory. All other parts are loaded on request if parts of the 3D point cloud in lower tree levels of the tree are required to visualize details. The calculation time depends mainly on available main memory and the speed of the external memory. Our test system with 4 GB main memory and a 5.400 rpm hard disk needs 14 hours to transform a dataset with 5 billion points (73 gigabyte) into the tree structure.

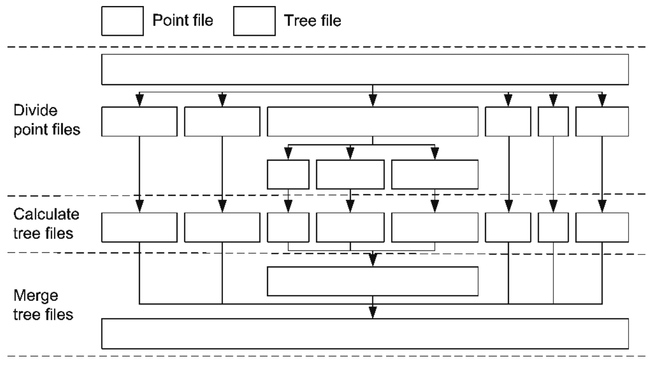


Figure 5: *Generation of an octree for a dataset that does not fit into main memory. The point file is divided depending on octree cells until a calculation in the main memory is possible. The resulting sub tree files are merged until one tree file with the entire dataset results.*

3.2 Rendering Strategies

The rendering algorithm is designed to fulfill two goals: the first purpose is the point throughput to render as many points as possible. This is necessary to present large parts of the 3D point cloud with a high resolution. The second one is to guarantee high frame rates during interaction to enable interactive exploration of the massive 3D point cloud. The application is designed for a user group that has knowledge about the characteristics of LiDAR data. The main aim of the visualization is to survey the captured LiDAR data. A fast evaluation of the data is important to guaranty a defined level of accuracy before the further processing of the data starts. The prevention of holes in the surface [Park et al. 2005] of the scanned target would improve the display quality for planar areas of the 3D point cloud but it would not represent the exact copy of the scanned target area for non-planar parts of the dataset. It would also require additional storage, preprocessing and rendering effort and is more important for surface reconstruction algorithms [Wang and Oliveira 2007]. For this reason we did not realized the prevention of holes in surfaces and other techniques to improve the display quality, e.g., illumination and shadows.

To fulfill our requirements the projected point spacing (PPS) of an octree node is used as quantity to render the nodes of the tree structure. The bounding sphere of the node is projected into the screen space and the number of covered pixel is calculated. The approximation of the covered screen space with a bounding sphere is not accurate in case of large planar surfaces, because the sphere covers more space than the point data in general. This is particularly disadvantageous if a planar surface is viewed from the side and it leads to more displayed points than necessary. This disadvantage is compensated through the fast culling of spheres against the view frustum. Other primitives, such as bounding boxes and oriented disks for the abstraction of nodes, enable a more precise estimation but are much more expensive to cull against the frustum. Tests with our rendering system have shown that culling a sphere against the view frustum is about three times faster than culling a box. Hence a deeper tree traversal with a constant rendering budget is possible. The memory need for storing a sphere is lower than storing a bounding box. This allows the algorithm to keep more nodes in main memory.

3.2.1 Optimization with the Rendering Front

The PPS threshold for displaying nodes is adapted dynamically depending on the user input and available hardware resources. Traditionally, a depth-first traversal of the hierarchy is performed until the PPS threshold is reached [Wand et al. 2007]. The traversal of the tree is time-consuming and, if less or no interaction or navigation is performed, a high frame-to-frame coherence is given. To exploit the frame-to-frame coherence all tree nodes selected for rendering are kept per frame. These nodes are stored in a list, called *rendering front*, which cuts the tree in exactly two parts (Fig. 6). All nodes above the rendering front have a projected point spacing (PPS) larger than the PPS threshold, and all nodes in the rendering front have a lower or equal PPS (Fig. 6). The rendering front is used for the next frame to check the validity of the nodes. If the projected size of a node is larger than the threshold, it can be replaced with its child nodes (Fig. 6 (a)). On the other hand, if the size of a parent node in the rendering front is below the PPS threshold, all nodes with this parent can be replaced by the parent node (Fig. 6 (b)). Each node in the rendering front is rendered with its position and color. If leaf nodes in the tree are reached by the rendering front, all points in the leaf node are rendered.

To achieve nearly constant frame rates as well as smooth refinements of the displayed 3D point cloud, we define a restriction for rendering front updates to estimate and limit the effort for each frame. The expansion of the rendering front is time consuming, because one node is removed and up to eight nodes are added. If the limit for expansions is reached, the remaining nodes in the rendering front are rendered without checking the PPS threshold, because selection of nodes for the current frame should be completed as fast as possible. In the next frame the expansions of the rendering front begins right behind the last expansions of the previous frame. Many updates of the rendering front are needed when the camera position changes rapidly, e.g., when turning the 3D point cloud or zooming into details. The budget for rendering front updates estimates depending on the number of expansions and the time to traverse the rendering front in the prior frame.

Our initial threshold for the projected size of nodes is approximately 60 pixels at the beginning of the visualization process. If the limit for rendering front updates is not reached, the threshold is decreased from frame to frame, otherwise retained. This leads to a smooth update of the entire displayed 3D point cloud with constant frame rates. While interacting with the 3D point cloud higher frame rates are desirable and the PPS threshold is increased until high frame rates can be assured. Figure 8 shows a LiDAR dataset displayed with different PPS thresholds. High thresholds lead to a less amount of points with holes in the displayed dataset. These holes can be compensated with larger point primitives. In each frame visibility culling is performed for each node to traverse and render only parts of the 3D point cloud that are inside or intersect the view frustum.

Another limiting value for the algorithm is the capacity of the GPU

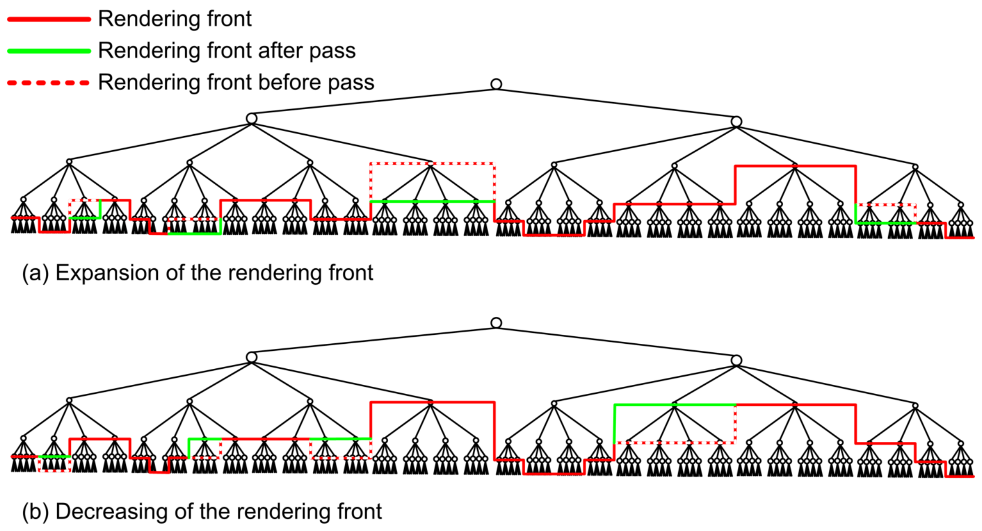

Rendering front
Rendering front after pass
Rendering front before pass
(a) Expansion of the rendering front
(b) Decreasing of the rendering front

Figure 6: *Illustration of the tree structure with the corresponding rendering front for one frame. In (a) the rendering front is expanded at 11 nodes because their PPS does not fulfill the PPS threshold; (b) illustrates the decreasing of the rendering front by replacing nodes by their parents.*

memory. Widely used graphic cards with 512MB memory can only store about 30 million points with a float precision of 4 bytes. Each point needs 16 bytes, 12 for the position and 4 bytes for the color. For that reason the algorithm does not expand the rendering front in case a maximum number of points in the rendering front is reached. This threshold depends on the available hardware resources.

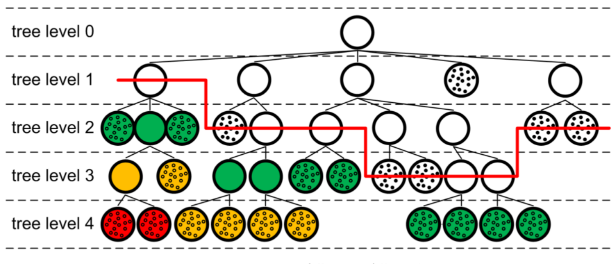
During the evaluation along the rendering front points are selected for rendering and transferred to the GPU. These points are directly streamed into the GPU memory to prevent blocking of GPU and CPU. There is no delay after evaluation of the rendering front and the transfer of point data to the GPU memory. Transfering a vertex buffer object with all points to the GPU after evaluation of the rendering front leads to 40 percent lower frame rates.

3.2.2 Out-of-Core Strategies

For massive 3D point clouds the tree structure does not fit into main memory. For this reason only relevant parts of the tree are loaded for rendering. Initially the first few levels of the tree are transferred from external memory into main memory. Nodes in the last loaded tree levels store information about the location of child nodes that are only available on external memory. If the rendering algorithm reaches inner nodes without child nodes in main memory, then a reload-task starts. This task is executed on separate CPU cores with lower priority to prevent disturbing or blocking the rendering algorithm. The task manager handles all node reload requests and attaches the nodes to the corresponding parents in the tree. Due to slower access times of external memory nodes are available for rendering a few frames after the request in regular. This duration depends on the external memory access times and the currently active reload tasks. Probably child nodes of the loaded nodes will be needed a few frames later, so all available child nodes of the initial requested node are loaded too. If the user interacts with the 3D point cloud some node reload requests can become unnecessary. Reasons can be that they are outside the frustum or too far away, and the parent detail level is sufficient. For that reason all reload requests are canceled in case of user interaction.

3.2.3 Memory Release Strategies

A fixed memory budget to prevent running out of main memory exists for the entire data structure. If this memory limit is reached, no more reloads can be performed, and the algorithm has to release memory resources by deleting unused parts of the tree structure. To decide, which nodes can be removed from the main memory, the rendering front is used: all nodes in the rendering front were rendered in the last frame. They serve as good indicator to approximate, which parts of the tree can become important for the rendering algorithm in the next frames. The possibility that a node is needed in the next frame decreases with the number of level of the node below the rendering front. All nodes that are located far enough under the rendering front are removed from main memory to release resources. This strategy must ensure that the algorithm never runs out of memory and additionally that not too much tree nodes are deleted. For that reason the rendering front is evaluated in two passes. In the first pass all nodes are counted depending on the distance to the rendering front, as shown in Fig. 7. After the count pass, the number of nodes for each level below the rendering front is known. For each level the memory assignment can be estimated and it is possible to decide how much distance levels should be selected to delete nodes.



Distance	#Nodes	#Leafs
🟢 1	3	8
🟡 2	1	5
🔴 3	0	2

Figure 7: *Illustration of the strategy to release memory. In the first iteration nodes are counted (table) while in the second iteration nodes with the highest distance are deleted from main memory.*

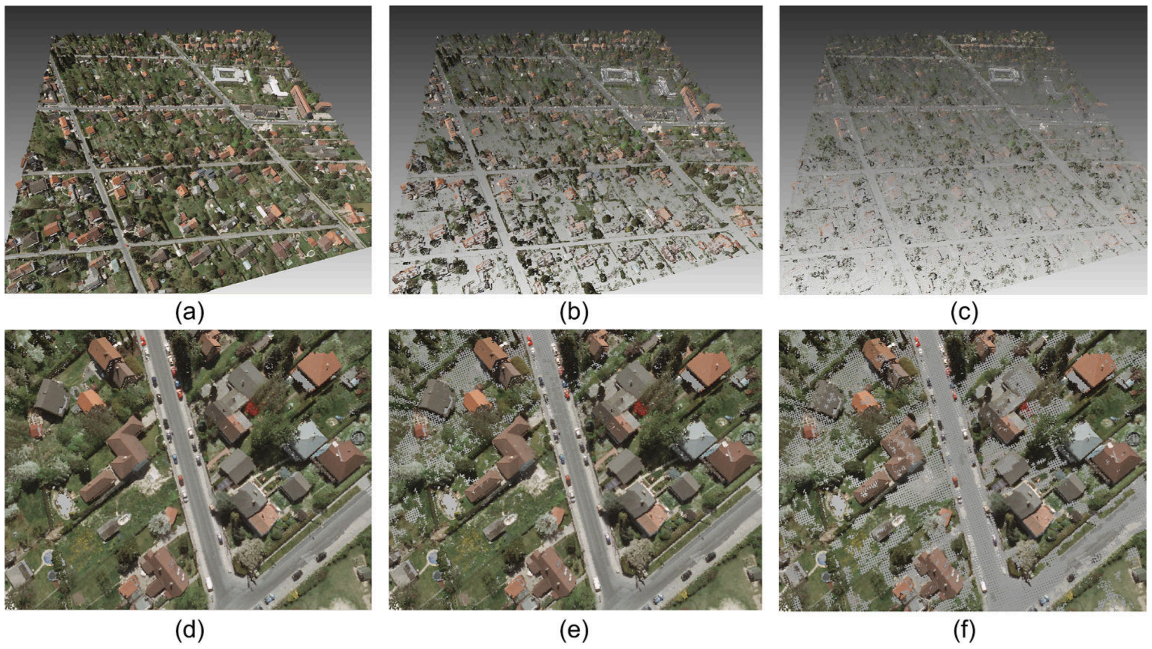


Figure 8: *Colored 3D point cloud with 20 million points captured with an airborne laser scanner. The 3D point clouds are renderd with differnet PPS thresholds, 5 pixel/10M points (a), 100 pixel/1.5M points (b), 200 pixel/0.5M points (c), 5 pixel/0.9M points(d), 100 pixel/0.6M points (e), and 200 pixel/0.5M points (f).*

4 Results

Our implementation for rendering massive 3D point clouds is based on C++ and OpenGL. We performed tests with different datasets from airborne and terrestrial laser scans. The largest dataset shown in Fig. 1 is a LiDAR 3D point cloud of the city of Berlin. The point density is 5-15 points per square meter; the entire dataset contains about 5 billion points. The size of the input data is 150 GB for the uncompressed LiDAR data and 80 GB for the aerial images. After the standardization and coloring process the 3D point cloud needs 73 GB in a binary format. The preprocessing to build the tree structure with a size of 78 GB takes 14 hours. The behavior of the rendering algorithm for different cases is illustrated in Fig. 9 and 10.

Figure 9 shows the behavior of the rendering front during exploration of the entire 3D point cloud. The diagram shows the sequence of the rendered points, the rendering front length and the PPS threshold values for interaction and a constant view position. If no interaction is performed, the rendering front is extended until a fixed PPS threshold is reached or the GPU memory is full. While turning around the 3D point cloud the PPS threshold is increased to a fixed value to enable high frame rates. This leads to an almost constant length of the rendering front. The fixed PPS threshold for interaction is 60 pixel for our dataset. If it is decreased, the number of rendered points increases, because the rendering front can be extended. If the PPS threshold is increased due to navigation, the rendering front is decreased as shown in Fig. 6 (b). The almost identical values for the length of the rendering front and the number of rendered points can be justified with the high location of the rendering front in the tree. Nearly all nodes in the rendering front are inner nodes, which are rendered with one point.

In Fig. 10 the behavior for zooming is illustrated. If only a small part of the 3D point cloud is inside the view frustum the PPS threshold can be decreased fast, because only a small part of the rendering front has to be updated. The maximal resolution of the 3D point cloud is also reached, and more leaf nodes instead of inner nodes of the tree are rendered. This leads to much more rendered points in comparison to the number of nodes in the rendering front. When zooming out, leaf nodes are removed from the rendering front and the number of displayed points decreases rapidly.

5 Conclusions and Future Work

We have presented a point-based rendering system that is designed to process and display massive 3D point clouds. The main advantages are rendering strategies that adapt automatically to available hardware resources such as main memory and GPU memory. We can achieve high frame rates during user interaction and also a high point density if no interaction is performed. Our main contribution is to show that it is possible with standard hardware to explore even 3D point clouds with billions of points in real-time. Because of the increasing precision of scanning devices the size of datasets will most likely grow dramatically in the future. To afford the exploration and to control quality of 3D point clouds an out-of-core visualization is necessary. Our approach enables a fast preprocessing and rendering of LiDAR data. It is well suited also for small datasets to get a good overview of the collected data. In our future work we want to integrate multi-core strategies into the rendering system to enable a faster evaluation of the tree structure on separate CPU cores to accelerate the point selection per frame.

Acknowledgements

This work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the InnoProfile research group 3D Geoinformation (www.3dgi.de). We would like to thank virtualcitySYSTEMS for providing LiDAR datasets and aerial images.

References

ABDELHAFIZ, A., AND NIEMEIER, W. 2006. Developed technique for automatic point cloud texturing using multi images applied to a complex site. *IAPRS XXXVI*, 1–7.

ABDELHAFIZ, A., RIEDEL, B., AND NIEMEIER, W. 2005. Towards a 3d true colored space by the fusion of laser scanner point cloud and digital photos. *International workshop 3DArch*.

BETTIO, F., GOBBETTI, E., MARTON, F., TINTI, A., MERELLA, E., AND COMBET, R. 2009. A point-based system for local and remote exploration of dense 3D scanned models. *The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST*.

BOTSCH, M., AND KOBBELT, L. 2003. High-quality point-based rendering on modern GPUs. *Pacific Conference on Computer Graphics and Applications*, 335–343.

BOTSCH, M., HORNUNG, A., ZWICKER, M., AND KOBBELT, L. 2005. High-quality surface splatting on today's GPUs. *Eurographics Symposium on Point-Based Graphics*, 17–141.

BOUBEKEUR, T., DUGUET, F., AND SCHLICK, C. 2005. Rapid Visualization of Large Point-Based Surfaces. *The 6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST*.

CALLIERI, M., PONCHIO, F., CIGNONI, P., AND SCOPIGNO, R. 2008. Virtual Inspector: a flexible visualizer for dense 3D scanned models. *IEEE Computer Graphics and Applications 28*, 44–55.

DACHSBACHER, C., VOGELGSANG, C., AND STAMMINGER, M. 2003. Sequential point trees. *ACM Transactions on Graphics 22*, 657–662.

DEY, T., LI, G., AND SUN, J. 2005. Normal estimation for point clouds: A comparison study for a Voronoi based method. *Eurographics Symposium on Point-Based Graphics*, 39–46.

GOBBETTI, E., KASIK, D., AND YOON, S. 2008. Technical strategies for massive model visualization. *ACM Solid and Physical Modeling Symposium*, 405–415.

HOPPE, H. 1996. Progressive meshes. *ACM SIGGRAPH*, 99–108.

ISENBURG, M., AND LINDSTROM, P. 2005. Streaming meshes. In *Visualization*, 231–238.

KREYLOS, O., BAWDEN, G. W., AND KELLOGG, L. H. 2008. Immersive Visualization and Analysis of LiDAR Data. *International Symposium on Advances in Visual Computing*, 846–855.

LEVOY, M., AND RUSINKIEWICZ, S. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. *ACM SIGGRAPH*, 343–352.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. *ACM SIGGRAPH*, 259–262.

MARTON, F., AND GOBBETTI, E. 2004. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics 28*, 815–826.

PAJAROLA, R., SAINZ, M., AND LARIO, R. 2005. XSplat: External Memory Multiresolution Point Visualization. *IASTED Visualization, Imaging and Image Processing*, 628–633.

PAJAROLA, R. 2005. Stream-Processing Points. *IEEE Visualization*, 239–246.

PARK, S., GUO, X., SHIN, H., AND QIN, H. 2005. Shape and appearance repair for incomplete point surfaces. In *International Conference on Computer Vision*, 1260–1267.

PAULY, M., KEISER, R., KOBBELT, L., AND GROSS, M. H. 2003. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics 22*, 641–650.

PAULY, M., KOBBELT, L. P., AND GROSS, M. 2006. Point-based multiscale surface representation. *ACM Transactions on Graphics 25*, 177–193.

PFISTER, H., ZWICKER, M., BAAR, J. V., AND GROSS, M. 2000. Surfels: Surface elements as rendering primitives. *ACM SIGGRAPH*, 335–342.

POULLIS, C., AND YOU, S. 2009. Photorealistic large-scale urban city model reconstruction. *IEEE transactions on visualization and computer graphics 15*, 654–69.

SAINZ, M., PAJAROLA, R., AND LARIO, R. 2004. Points reloaded: Point-based rendering revisited. *Symposium on Point-Based Graphics*, 121–128.

SCHEIBLAUER, C., ZIMMERMANN, N., AND WIMMER, M. 2009. Interactive Domitilla Catacomb Exploration. *Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST*.

WAHL, R., GUTHE, M., AND KLEIN, R. 2005. Identifying planes in point-clouds for efficient hybrid rendering. *The 13th Pacific Conference on Computer Graphics and Applications*, 1–8.

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2004. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. *Eurographics Symposium on Rendering*.

WAND, M., BERNER, A., BOKELOH, M., FLECK, A., HOFFMANN, M., JENKE, P., MAIER, B., STANEKER, D., AND SCHILLING, A. 2007. Interactive Editing of Large Point Clouds. *Eurographics Symposium on Point-Based Graphics*, 37–46.

WANG, J., AND OLIVEIRA, M. 2007. Filling holes on locally smooth surfaces reconstructed from point clouds. *Image and Vision Computing 25*, 103–113.

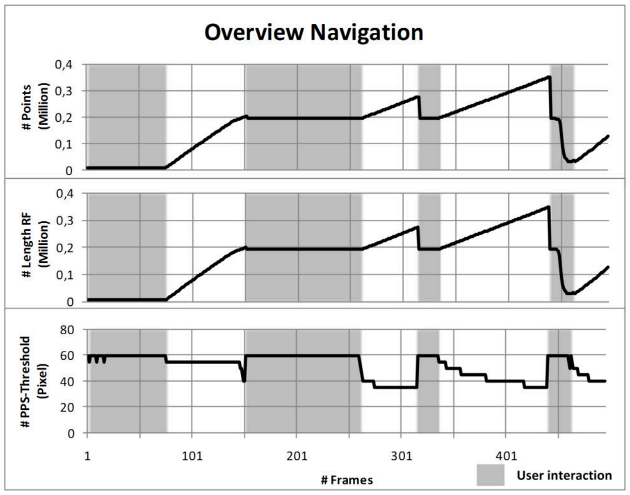
WIMMER, M., AND SCHEIBLAUER, C. 2006. Instant Points: Fast Rendering of Unprocessed Point Clouds. *Eurographics Symposium on Point-Based Graphics*.

XU, H., AND CHEN, B. 2004. Stylized rendering of 3D scanned real world environments. *3rd international symposium on Non-photorealistic animation and rendering - NPAR*, 25–34.

ZHAO, W., NISTER, D., AND HSU, S. 2005. Alignment of continuous video onto 3D point clouds. *IEEE transactions on pattern analysis and machine intelligence 27*, 1305–1318.

ZWICKER, M., PFISTE, H., BAAR, J. V., AND GROSS, M. H. 2001. Surface Splatting. *ACM SIGGRAPH*, 371–378.

ZWICKER, M., PAULY, M., KNOLL, O., AND GROSS, M. 2002. Pointshop 3D: an interactive system for point-based surface editing. *ACM Transactions on Graphics*, 322–329.



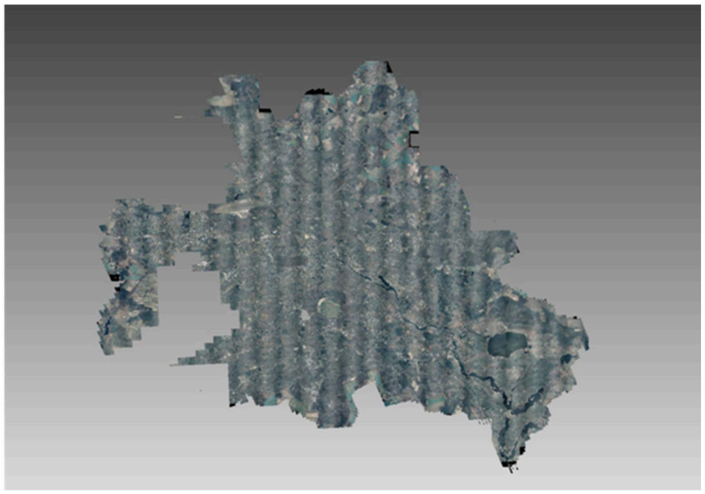
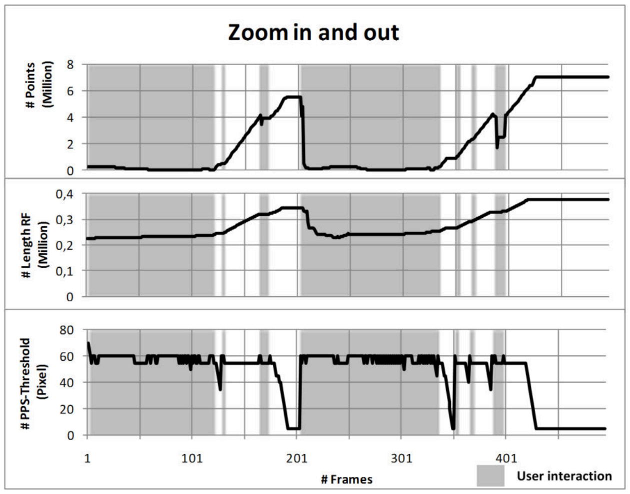


Figure 9: *Behavior of the rendering algorithm during navigation to get an overview.*



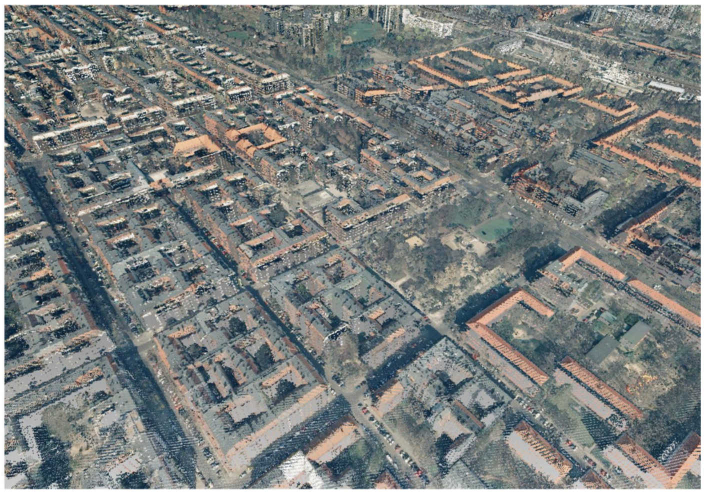


Figure 10: *Behavior of the rendering algorithm while zooming in and zooming out.*